Published by ModShop
May 16, 2011 (last update: Dec 24, 2014)

How to Make a Game

Score: 4.2/5 (1469 votes)

It has been recently brought to my attention that many people ask about game development, yet there aren't many articles on the topic. I've decided to shed some light on the general process of developing a game from start to finish. Keep in mind that that this is primarily an overview and: A. Won't apply verbatim to all projects. B. Is not a complete, step by step guide to a finished result. You will still need to figure out quite a bit of stuff on your own to have a game.

Step 1: Choose Your Game Library

Unless you want to write your own library for all the nitty gritty graphics/sound programming, you will probably want to get a game library. There are many game libraries out there, but they all offer the same base functionality

Features that you want your library to have:

- A way to load and render images
- A way to load and play audio
- Basic image manipulation (rotation, etc)
- Primitive drawing capabilities(circles, lines, rectangles, dots, etc)
- Ability to render text
- Ability to track time and wait
- Ability to create and control threads (is nice, but not required)

Some game libraries include:

- Simple Fast Multi-Media Library (SFML): http://www.sfml-dev.org/
- Simple DirectMedia Layer (SDL): http://www.libsdl.org/
- Allegro: http://www.allegro.cc/
- OpenGL (Rendering only, however, there are wrapper libraries like AllegroGL): http://www.opengl.org/
- DirectX (Windows only): http://msdn.microsoft.com/en-us/directx/
- Irrlicht (3d library): http://irrlicht.sourceforge.net/

Step 2: Define the Concept

All games start here, merely ideas in someone's head.

First, come up with an idea for a game. Once you have a simple idea, expand on it. For example, if it is a board game, what is the objective/How do you win? What will the rules be like? etc. If your game will have characters or a story, create them. Make sure you have a pretty well-defined concept of what your game will be when it's finished. The more complex the game, the better you should plan it out in the beginning so you don't have to worry about the game itself while your coding. Keep in mind that your game WILL evolve as you create it.

Step 3: Plan Your Engine

If you're making a board game or basic arcade equivalent you can skip this entirely and simply program your game. For more complex games, however, you may want to look into using a pre-made engine, or writing your own "engine". What even is a game engine you ask? While they vary widely in structure and overall functionality, you can think of a game engine as a super-powered library that provides higher level functionality such as physics, resource handling and game entity management. Whether you choose to use an existng engine or create your own, so to speak, is up to you and depends on how much programming you actually want to do. Using a pre-made engine will simplify your job as a programmer to scripting gameplay/events more than anything else.

Why did I say plan rather than choose? Well, odds are you're not making the next Elder Scrolls, and therefor, can create your own "engine" of sorts. Keep in mind that you will not be creating the next Unreal Engine, and most of the code you write and intend to be reusable (as is the point of an engine) will end up being so intertwined with your game logic that it is impossible to reuse easily. With this in mind, don't worry if parts of your "engine" rely on code specific to the game, this is just going to happen. Instead of focusing on making a completely reusable, super robust framework, focus on making sure the code is readable, organized and functional. Focus first on making the game, then try to create portable modules. If you absolutely must write something useful and reusable, resource managers and other various utility classes are good starting points.

Step 4: Write Your Engine (if you're making your own)

Now its time to actually start writing your engine, provided this is the route you chose. This doesn't necessarily mean the game itself, but rather, core rendering, physics, and file handling; essentially the functions and classes that will be used to construct your game. Simple games won't really require much of a framework, and can just be programmed using your game library directly. One of the most important, and most neglected, components of larger games is the resource manager. The resource manager is (presumably) a class that is responsible for loading resources (think graphics and sound), ensuring that resources are loaded only once, and unloading resources when they are no longer needed. RAM isn't infinite, so if your game is loading a separate copy of the same image for every piece of grass in the universe, you're going to have a bad time. See an excellent resource manager below by **Xander314**.

Resource Manager by **Xander314**

```
/*
 ResourceManagerB.hpp - Generic template resource manager

 (C) Alexander Thorne (SFML Coder) 2011
 <a href="http://sfmlcoder.wordpress.com/">http://sfmlcoder.wordpress.com/</a>

 Manages loading and unloading of a resource type specified by a
 template argument.

************************************************************************/

#include <map>
#include <string>
#include <exception>

typedef const std::string URI;

// exceptions
namespace Exceptions {

        // thrown if user requests a resource URI not present in the manager's list
        class URINotFound : public std::runtime_error
        {
        public:
                URINotFound(const std::string& Message = "The specified URI was not
                        : runtime_error(Message) { }
        };

        // thrown if a resource allocation fails
        class BadResourceAllocation : public std::runtime_error {
        public:
                BadResourceAllocation(const std::string& Message = "Failed to alloca
                        : runtime_error(Message) {}
        };
}

template <class Resource> class ResourceManagerB {
```

```
38          typedef std::pair<URI, Resource*> ResourcePair;
39          typedef std::map<URI, Resource*> ResourceList;
40
41          // the list of the manager's resources
42          ResourceList Resources;
43  public:
44          ~ResourceManagerB() { UnloadAll(); }
45
46          // Load a resource with the specified URI
47          // the URI could represent, e.g, a filename
48          URI& Load(URI& Uri);
49          // unload a resource with the specified URI
50          void Unload(URI& Uri);
51          // unload all resources
52          void UnloadAll();
53
54          // get a pointer to a resource
55          Resource* GetPtr(URI& Uri);
56          // get a reference to a resource
57          Resource& Get(URI& Uri);
58  };
59
60  template <class Resource>
61  URI& ResourceManagerB<Resource>::Load(URI& Uri)
62  {
63          // check if resource URI is already in list
64          // and if it is, we do no more
65          if (Resources.find(Uri) == Resources.end())
66          {
67                  // try to allocate the resource
68                  // NB: if the Resource template argument does not have a
69                  // constructor accepting a const std::std::string, then this
70                  // line will cause a compiler error
71                  Resource* temp = new (std::nothrow) Resource(Uri);
72                  // check if the resource failed to be allocated
73                  // std::nothrow means that if allocation failed
74                  // temp will be 0
75                  if (!temp)
76                          throw Exceptions::BadResourceAllocation();
77                  // add the resource and it's URI to the manager's list
78                  Resources.insert(ResourcePair(Uri, temp));
79          }
80          return Uri;
81  }
82
83  template <class Resource>
84  void ResourceManagerB<Resource>::Unload(URI& Uri)
85  {
86          // try to find the specified URI in the list
87          ResourceList::const_iterator itr = Resources.find(Uri);
88          // if it is found...
89          if (itr != Resources.end())
90          {
91                  // ... deallocate it
92                  delete itr->second;
```

```
                // then remove it from the list
                Resources.erase(Uri);
        }
}

template <class Resource>
void ResourceManagerB<Resource>::UnloadAll()
{
        // iterate through every element of the resource list
        ResourceList::iterator itr;
        for (itr = Resources.begin(); itr != Resources.end(); itr++)
                // delete each resource
                delete itr->second;
        // finally, clear the list
        Resources.clear();
}

template <class Resource>
Resource* ResourceManagerB<Resource>::GetPtr(URI& Uri)
{
        // find the specified URI in the list
        ResourceList::const_iterator itr;
        // if it is there...
        if ((itr = Resources.find(Uri)) != Resources.end())
                // ... return a pointer to the corresponding resource
                return itr->second;
        // ... else return 0
        return 0;
}

template <class Resource>
Resource& ResourceManagerB<Resource>::Get(URI& Uri)
{
        // get a pointer to the resource
        Resource* temp = GetPtr(Uri);
        // if the resource was found...
        if (temp)
                // ... dereference the pointer to return a reference
                // to the resource
                return *temp;
        else
                // ... else throw an exception to notify the caller that
                // the resource was not found
                throw Exceptions::URINotFound();
}
```

Another important aspect of your engine/framework is the interface. When you're writing the logic of the game itself it shouldn't take you 4 hours to write the main game loop as you search through the hundreds of update functions trying to figure out which ones you actually need. Keep it simple and concise. If you're able to update all game logic with one or two function calls and render the

scene with one or two more you're on the right track. Taking advantage of object oriented principles, such as inheritance and pure virtual base classes (think ***interfaces***) is a great way to create a framework with a sound structure.

For example, a base class of all game objects could be defined as such:

```
class GameObject
{
public:
    virtual ~GameObject()=0;

    virtual Vector2f getPosition();

    virtual bool interact(Object* o);

    virtual void draw(); //highly library dependent
};
```

With all subclasses now being held to this interface it is possible to have one holding entity that can easily store and manage any and every object you define regardless of what the object actually is. As you learn and program more and more you'll find more ways of using the various features of your language of choice to your advantage.

Step 5: Media (audio and graphics)

By now you've hopefully at least thought about what you want the game to actually look like, and maybe you already have a set of media to work with. If you're anything like me, however, you got so excited and caught up in whatever "beautiful design" you've thought up that by the time you get it to the point of testing you don't have a single image for your creation to make dance on the screen. Now is a great time to start getting the resources that you need. If you're artistically inclined, that's awesome. If not, don't worry, hope is not lost. Massive quantities of free graphics and sound effects are just a Google search away. Audacity and GIMP are indispensable tools for editing whatever you get or create.

Step 6: Write Your Game

Once you have an engine chosen or your own framework to use you can get around to actually writing the game logic itself. Ideally you've read through this article in its entirety at least once

before spending countless hours that you'll never get back on creating an "engine" that oversteps its role enough to be virtually unusable but not enough to be independently functional. Your framework should provide a base that structures object interaction (but doesn't necessarily define it) and handles all rendering and other low level details, such as physics. The game logic itself will define object interaction (by defining a subclass of GameObject for example), game rules (such as what constitutes winning or losing), the initial state of the game (which map is loaded first, what items you start with, etc) and will contain the ***main game loop***.

What on earth is the main game loop? Simply put: it's a loop, the main loop. Think of what gets repeated continually while the game is being played, those are the things contained in this mysterious loop. For example, every iteration the game should update all of the objects and then draw them all to the screen. Beyond updating and drawing, the main loop will also probably be responsible for timing. A game that updates too much will appear to be incredibly fast to the user, and most likely will be too hard. Think pong at the speed of light. Ideally this loop will use the framework you created earlier and will be quite simple itself. See the example below:

Game Loop:

```
while (!Game.playerLost())
{
    world.update(); //assume this world object owns all of the GameObjects and updat

    screen.clear();
    world.draw(screen);
    screen.display();

    ensureProperFPS(); //just a placeholder, put the actual timing logic right here
}
```

Step 7: Take something from it

The main reason I talk about creating a framework separate from the game logic is for you to learn to write reusable code. Then I told you not to worry about making it actually reusable and instead focus on making the game. I stand to that, one of the main reasons beginners give up on projects is that they spend vast amounts of time and effort trying to "write an engine" for their game, but they don't really know yet what a good engine even includes or a structure/interface that will actually work. After wasting all of that time they have nothing to show, then get discouraged and quit as a result. By focusing on writing the game first, and reusable code second, you will ideally end up with something that you can see. A tangible reward for your effort and a reason to keep working harder.

Now that you have a playable game that you're happy with you can try and make portable modules out of the game code. Did you write an awesome resource manager or fantastic class for handling keyboard input? Try and make them completely portable so you can just copy over the source files and use them right "out of the box" in other projects. If you want to start completely fresh in your next project, that's fine as well. You don't have to literally take code from a project to have taken something from it. So long as you learned something in the process it was all worthwhile.

Step 8: Package and Distribute

After all of that work you'll probably want people to actually play your game! Package all of the required files into a zip file, compressed archive or executable installer and send it to everyone!

Tips:

I've learned many things about making games, some things the hard way. Here are some things that you should do:

- First, stay organized! You should have a good organizational system for everything; your code, your media, your documentation, etc. Folders exist for a reason, use them!

- Also, try to keep your code clean and readable. Give functions meaningful names and keep everything as simple as possible

- Document! I never really talked about it in the article, but document everything! Document the format for all of your data files and document what all of your functions and classes do. You have no idea how much time this saves and how many headaches it prevents until you do it

- Start small. Don't try and create the next Pokemon game on your first go. Start with small, manageable projects and expand your goals in proportion with your skills. Trying to tackle a project over your head will only discourage you

- Eye on the prize! One of the biggest downfalls to one too many projects of mine was me getting hung up on tiny details while neglecting the bigger picture. Yes, my clouds, rain and thunder, footprints and fog effects were all beautiful, but I never ended up with a finished game. You can make it pretty later, make the game first!

- Have fun! The point of games is to have fun, and making them can be fun, too. It's always easier to accomplish a task if you enjoy yourself while working on it. Of course there will be times when you get frustrated, but if you find yourself getting overly angry, take a break! Taking a walk and thinking about something else is often the best solution to a problem. When you return you'll have a fresh start, which can allow you to find a solution that you may not have thought of while on your previous train of thought.

Some starting tips from chrisname:

You don't need to work that hard. What you need to do, is go through a programming tutorial (the one on this website for example). Don't do too much in a day, or you will get bored and unmotivated. Don't set a goal based on time, that doesn't work. You'll forget alot of what you learn if you stop halfway through a lesson. Work through the tutorial on this website (http://cplusplus.com/doc/tutorial/). Aim to get through two lessons a day. Don't stop partway through a lesson (unless it's for a short break, that's a good idea) and don't do too much in one go, or you simply won't remember it. I recommend reading and copying out each example (not copy and paste; type it yourself, this will help you to understand what you are doing), compiling it, seeing what it does when you run it and modifying things to see what changes. I also recommend you look at other people's code (one of the things that has helped me is to take other people's broken code and try to fix it, although don't get too hung up on this because it is hard to read other people's code when you first start out). When you are reading, try to rephrase things: "If you can't explain it simply, you don't understand it well enough." (Albert Einstein).

Once you've gone through this tutorial, and maybe some others (I read about three different tutorials because it was useful to have things said in a different way - I find having something explained in two different ways is useful for understanding and remembering it), you could read through the tutorials for SFML (http://sfml-dev.org/tutorials/1.6/). Learning SFML will teach you to make 2D games. I'd also recommend learning SDL (http://lazyfoo.net/SDL_tutorials/index.php) because lots of games use it and you will most probably come across it eventually.

After that, you should get into OpenGL programming if you want to make 3D games. SFML makes this very easy, and the SFML tutorial includes a tutorial for using OpenGL. For OpenGL, perhaps someone here can recommend you a book or tutorial.

Throughout all this you should remember that it is important to pace yourself. Don't try to absorb too much at once or you'll forget lots of it. And don't stay up until 3am when you have an exam the day after next...

© cplusplus.com, 2000-2023 - All rights reserved - ***v3.3.3***